CHAPTER 1
INTRODUCTION

The faster you make a computer run, the harder it is to envision how to make it run still faster. It is no longer sufficient to just make the architecture process instructions faster. You must make more things happen at once—that is, in parallel. This feels like cheating—the computer is not running faster—it is simply getting more done at once.

The IA-64 architecture employs a new parallel computer architecture called *explicitly parallel instruction computing* (EPIC). Intel Corporation's first hardware implementation of this architecture is the Itanium™ processor. EPIC offers many new features to increase performance, but the most important are explicit instruction level parallelism (ILP), speculation, predication, and a large register file. Each of these makes a computer run faster.

But what do you do with all this parallelism? The traditional problem is that we never have enough work ready to keep a machine fully busy. If we have all this power, the key is to not throw it away by letting it go unused. Anytime the processor is ready to do six more things, we don't want to give it two things to do and waste 66% of the processing power.

A breakthrough happens when you decide to stop worrying about only doing things we must. The approach is to ask the machine speculatively to do four more things. That is, things are picked that might be needed in the future, but right now we just aren't sure which of the four it will be. Let us say that the four extra things turn out to be needed only half the time. We are still better off than if we had asked the processor to do nothing. Half of the time the work is done by the time it is needed.

In fact, this is the promise of EPIC. As machines get wider and wider, we just speculate more and more. The goal is to compute results before they are needed, so that when a program needs them, the answers are already there. Nothing is faster than something that is already done!

In a traditional microprocessor, doing things in advance can create trouble. For instance, if A+B is computed too early, the values of A or B might change and make the original result useless. The sum of A and B must be recomputed. Because of these problems, the architecture requires

special features for speculation that make these exceptions efficient to detect and correct. Speculation is a big part of what EPIC offers, and it is offered in many forms. But, all the speculation features are simply in EPIC to allow us to compute things before they are needed, and before a check is made to ensure that it is not too early to do the work.

EPIC builds on the ability of a *very long instruction word* (VLIW) machine by allowing the selection of instructions to be executed in parallel to be done at compile time. For this reason, compilers may be the most important concept for EPIC. Compilers to harness the instruction level parallelism are very complex. IA-64's EPIC architecture is optimized for compiler writers, to simplify the task of writing the advanced compilers needed for EPIC. The goal of IA-64 architectural features, such as speculation, predication, and a large register file, is to simplify the problems that a compiler writer faces so that it is possible to write good compilers to use EPIC.

The first thing EPIC offers to compilers is explicit access to the parallelism—hence the term EPIC. This access is a bonus for the compiler writer and the microprocessor designer. Both jobs are simplified by opening up the communication between compiled code and the hardware. Once explicit parallelism is available, the challenge is to make full use of it. Speculative execution of instructions is the most important concept in EPIC for helping a compiler fill up the available parallelism. It is what makes it possible to believe that programs can be written to use the parallelism available. More importantly, speculative execution makes compiler writers believe they can compile programs into parallel code for the processor.

Predication is the second most important concept in EPIC for filling up the available parallelism. Practically every instruction that is executed is conditional, based on a predicate register. Each instruction either has an effect, or has no effect, based on the True or False condition in a predicate register. Branches in program control impede full use of parallelism. This predication of every instruction, with no extra run-time cost, allows the elimination of a branch, thereby increasing parallelism.

EPIC also offers very large register sets. A smaller register set limits performance. The processor has to shuffle data in and out of registers instead of doing the work required for the program. The registers in the IA-64 architecture even offer a feature called rotating registers, which reduces the need for register shuffling and code duplication when doing a type of loop level parallelism called software pipelining

Finally, EPIC acknowledges that the parallelism, which a compiler can find, is not consistent from clock to clock, and machine width is not fixed for every microprocessor design. The Itanium processor is six execution units wide, but cannot do every combination of six things at once. Therefore, EPIC allows code to express where the parallelism is, without forcing all code to be six wide forever. Future IA-64 microprocessors may be more flexible about what they can do in parallel or how many things can be done in parallel, or both. In earlier VLIW machines, inflexibility was a substantial problem in their designs. Width changed from generation to generation of the machine and the code compiled for one machine did not work on another. That is, they were not scalable. EPIC solves the scalability problem and code can execute across all IA-64 machines.

CONCEPTS LEADING TO EPIC

The two ways of making a computer go faster are: make it do each thing faster and make it do more things at once. Through these methods, a computer does more things in a given time and is recognized as being “faster.” Neither running fast, nor juggling lots of things at once is simple. A number of architectural innovations have driven the evolution from CISC microprocessors to EPIC-based processors and the resulting improvements in performance. This section explains these architectural capabilities.

PIPELINING

Consider a simple computer that executes one instruction, then another. Even this simple computer is complicated. The computer starts by reading the instruction from somewhere, interpreting it, and doing the required action. Often that action requires information (data) to be obtained (read) from somewhere, acted upon (perhaps added together) and the result to be written out. The computer ends up doing five things to carry out (execute) each instruction. They are:

READ INSTRUCTION
READ FIRST INPUT DATA
READ SECOND INPUT DATA
ACT ON DATA (ADD INPUTS TOGETHER)
WRITE ANSWER OUT

If the computer can read the next instruction at the same time as one of the other steps in the instruction execution process is taking place, instructions execute a little faster. This read cannot be done during a step that involves another read or write. For this reason, the read of the next instruction is usually performed overlapping with the act on data step. This architectural innovation is known as *pipelining*.

LOAD/STORE ARCHITECTURE

The simple computer used to demonstrate pipelining is in fact more complicated than it needs to be. If the rules are changed to eliminate the data read step, the computer can be simpler. The catch here is that an instruction is not what it used to be. In our original design, an instruction could get its data from anywhere, specifically, the memory subsystem. An instruction is no longer able to read values from two memory locations, ADD them, and write the sum to memory. Now, the data must be held in registers within the processor.

The architecture of the computer must be changed by adding instructions to read information in memory into registers and to write information in registers into memory. These additions are known as the load instruction and store instructions, respectively. Now separate instructions are available to load from memory to a register, add data in registers, or store the result in a register to memory, but only one operation can be done at a

time. The computer is now a *load/store machine*. Data accesses are made with load and store instructions, no longer as part of the execution of another instruction. That is, data can only be loaded from or stored to memory explicitly.

Load / store machines are easy to implement and generally run faster. Each instruction is easier to implement. The catch is that it now takes more instructions to do the same work. For instance, the complex instruction in our earlier example, which added the values in two memory locations and stores the result in memory, is implemented with four instructions:

```
LOAD INPUT 1
LOAD INPUT 2
ADD
STORE RESULT
```

The use of registers turns out to be efficient at reducing the number of memory loads and stores. If inputs already exist in registers or results can remain in registers, the need for memory accesses are eliminated. For instance, if the add operation in our example is actually an accumulate operation, input 2 and the result are located in the same register and the result is created by the add operation. Now the program sequence reduces to two instructions:

```
LOAD INPUT 1
ADD INPUT 1 TO INPUT 2
```

How does our new computer perform the operation of our earlier complex instruction example? Let us assume that it takes the load / store architecture 2 instructions to do the same work. A first estimate is that the computer needs to run twice as fast just to keep up with the old way of programming. However, our new computer has three steps instead of five for each instruction:

```
INSTRUCTION LOAD
ACT ON DATA FROM REGISTERS [the act may be a LOAD or a STORE]
PUT RESULT IN REGISTER
```

As shown in Fig. 1.1, instruction 1 is performed in three clocks.

The interesting twist now is that the “instruction load” step for the next instruction can be performed at the same time as the “put result in register” step of the prior instruction. This pipelining can be

accomplished because the computer is not busy talking to memory. In fact, if the “act on data” step is not a load or store that uses memory, the “instruction load” can be executed during this step. Figure 1.1 shows this overlapping of the execution of instructions 1 through 3. Now the three instructions are executed in five clocks. With these improvements, performance approaches one instruction per clock.

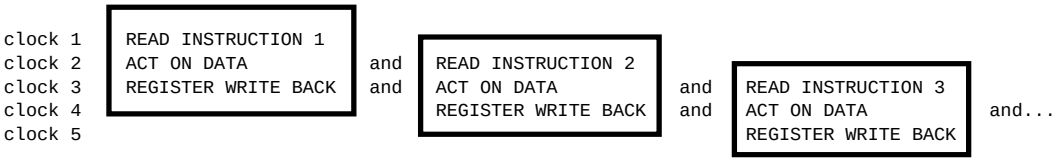

Figure 1.1. Pipelined Execution In A Load/Store Architecture

Notice that during clock 3 all three steps of instruction execution are performed. The result of instruction 1 is saved in a register, the data of instruction 2 is acted upon, and instruction 3 is loaded. We could say that this computer is doing one instruction per clock. However, if the “act on data from register” operation is a load or a store, the next “instruction load” has to be delayed one clock.

CACHE

Occurrence of load and store operations interferes with the pipelined execution of instructions. Like other problems, this one has a solution. Another enhancement to the architecture called caches is the solution. A cache is simply a small, fast memory that is located within a processor. Adding a few extra circuits, the processor uses the caches automatically without requiring the programmer to do anything other than read and write memory just as one would do without caches.

The cache memory allows many of the reads and writes for our example computer to happen at full speed by keeping copies of the value from parts of main memory in the cache. When the processor reads or writes and the data exist in the cache, a read hit has occurred and the data is accessed from the cache, not main memory. This memory access is completed in one clock. Delays still happen when the cache does not have a copy of the data. This condition is called a read miss. In this case, the data must be read from the main memory. Delays also happen if the cache is full of written data, and needs to make room for newly requested

data. In this way, we see that the existence of a cache alone does not assure that all memory accesses occur in one clock.

Caches actually have the ability to solve several problems. Today's microprocessors are faster than available memory chips. For this reason, multiple clocks always are required to access information located in main memory. These extra clocks reduce the performance of the computer. For our simple computer to keep running faster, loads and stores must run in one clock a high percentage of the time. Single-clock memory accesses are achieved for all data held in cache.

Whether you have a load/store architecture or not, lots of instructions use memory for data reads and writes. These loads and stores of data may interfere with the loading of instruction code in the "read instruction" step. Again, the impact is a decrease in pipelined execution. If we can avoid this conflict, our computer can truly feel like it is a one instruction per clock machine.

Two standard solutions have evolved for eliminating the conflict between instruction reads and data loads and stores. One approach is to have two memory systems, one for instructions and another for data, and connect them to the processor in a way that they do not interfere. This configuration is called *Harvard architecture.* A better solution is to use cache memory. The computer has just one main memory, but two caches—one for instructions and one for data. The code cache and the data cache are placed inside the processor in a way that they do not interfere. This architecture is known as a *von Neumann architecture*. This latter approach is preferred mostly because it requires only one set of connections to the main memory outside the processor. The reduced number of external connections is important because there is always be a limit on how many pins you can have on the processor.

Many instructions operate on two inputs and therefore require data to be loaded from two places. This last problem can also be resolve through the use of cache. The cache can be designed to retrieve two pieces of data at once. This organization is called a "multi-ported" data cache design.

To maximize performance, we will assume our simple computer has the benefit of a cache that eliminates multiple clocks for memory accesses, removes conflicts between instruction reads and data loads and stores, and provides multi-port access to memory. The performance achieved with this redesign is much better than that of our original complex instruction computer design. Cache design can get much more

complicated in order to be faster and faster, or to be able to serve multiple data in one clock. In this sense, caches, the same way as whole computers, can be designed to do things faster, and do more things at once.

BRANCH PREDICTION

One thing that computer programs do regularly is make a decision, and go to different pieces of code (instructions) depending on that decision. A branch instruction accomplishes this change in program control. If the result of the decision is to branch, we go to somewhere else to continue or if the decision is not to branch, we fall through the branch and execute the next instruction.

Branching is generally bad, because it interferes with our pipelined model of reading the next instruction while executing the last instruction. If that last instruction was the branch, we find ourselves in a dilemma. We do not know where to get the next instruction in time to fetch it while executing the prior instruction. One approach would be to wait until we know, before fetching the next instruction. This approach of waiting ensures that we get nowhere during the clocks that we wait. Fully pipelined execution is interrupted.

Two solutions emerged in processor designs. One approach is to guess where the next instruction is going to come from, and to fetch the next instruction from there. If we are correct, the pipeline remains full and parallel execution is maintained. If we are wrong, we need to figure that out and make things right later. The time that is lost recovering is known as a *branch-mispredict delay*. Predicting the place to get the next instruction is a form of simple speculation, which is a very important concept to the IA-64 architecture.

The other solution, which is more ingenious and worked better for a while, is called a *delay slot*. The idea is simple. You move the instruction immediately before the branch instruction to the line after the branch instruction. That is, you place it in the delay slot. Now a branch always must execute the instruction after the branch and then make the branch. In this way, the pipelining and parallel execution is maintained while the decision of the branch is made. This clever idea worked very well as long as your could figure out where the branch was going in the time the processor took to execute that single instruction in the delay slot, and as long as you could fill the delay slot with something useful. For a while,

this worked. But as time went on, more than one clock of delay was needed to be sure of the branch's destination and even if you added more delay slots, you had a harder and harder time filling them with useful work. So, the delay slots ended up just being delays, and the branch prediction scheme looked better and better. Eventually, all microprocessors migrated to branch prediction and the techniques to predict well have become more and more complex. The inability to predict the future (a branch) perfectly, every time, means that eliminating branches altogether seems like a great idea.

BRANCH ELIMINATION—PREDICATION

Once we know that branches cause problems, it is natural to want to eliminate them. The process by which branches are eliminated is known as *predication*. Consider a simple piece of C++ code:

```
if (a > b) {
      x = a
      z = 1
} else {
      x = b
      z = z + 1
}
```

If the processor has the ability to predicate an action, you can rewrite your program to avoid branching all together:

```
test = TRUE if (a > b), else FALSE
      if (test is TRUE) tmp1 = a
      if (test is FALSE) tmp1 = b
      x = tmp1
      if (test is TRUE) tmp2 = 1
      if (test is FALSE) tmp2 = z + 1
      z = tmp2
```

This predication capability is added to many microprocessors today, including the Intel® Pentium® III processors.

This feature turns out to be very hard to use, since the processor now executes a lot more instructions than were required for the original code with the branch. To pick the best code, you need to compute how often the branch is mis-predicted (has a penalty) versus correctly predicted (no penalty). The new code is clearly slower if the branches would have been predicted correctly, and might be faster only when they would have mis-predicted.

The concept of predication makes a lot more sense in EPIC, where it is combined with the ability to execute a lot of instructions at once (parallelism). In this case, the limited number of instructions inside the then branch (x=a and z=1) and the else branch (x=b and z=z+1) limit our performance. For code with a branch, a machine that can do six things at once can finish in 3 clocks plus any delays due to mis-predicted branches, for code with a branch:

```
clock 1  test = TRUE if (a > b), else FALSE
clock 2  if (!test) goto LABEL1
clock 3  x = a
   and      z = 1
   and      goto LABEL2
      LABEL1:
clock 3  x = b
   and      z = z + 1
      LABEL2:
```

Without a branch, the machine can do the operation in 2 clocks, with no branch delays:

```
clock 1 test = TRUE if (a > b), else FALSE
clock 2 x = a     only if test is TRUE
  and       x = b       only if test is FALSE
  and       z = 1    only test is TRUE
  and       z = z + 1   only test is FALSE
```

SUPERSCALAR

The predication example is all an excellent lead-in for the next topic: *parallelism*, that is, the ability to do more things at once. In our simple computer, fetching instructions while still executing previously fetched instructions is parallelism, in the sense of doing several things at once. However, we called this pipelining and not parallel execution. The reason is simple. With pipelining, each instruction is in a different part of its execution at any point in time. Each starts by itself and each ends by itself. If we start multiple instructions at the same time, we call it *parallel execution*. The net effect is similar; the computer is faster because it is getting more work done.

The simplest form of parallel execution is called *superscalar*. Scalar is a reference to doing one thing at a time, and superscalar means doing more than one thing at a time. The Pentium processor from Intel is a superscalar processor. Instead of starting one instruction at a time, the processor also starts the next two instructions at the same time whenever

possible. This not always possible. A common problem is that one instruction computes an answer that the next instruction uses as its input. Thus, the second instruction must wait for the first to be done before it can start. Naturally, such instructions cannot be done in parallel. A simple superscalar design like the Pentium processor is forced to wait for the first instruction to complete before doing the next instruction. After one instruction is done, the processor checks whether the next two can be run in parallel. This way, every clock the processor tries to start two instructions but falls back on starting just one when the two under consideration had some dependency on each other.

DYNAMIC EXECUTION

A variety of techniques can improve a simple superscalar design. The objective is to find a way to always execute instructions in parallel, even if the next two instructions have a dependency.

Intel® microprocessors based on the P6-microarchitecture employ a collection of these techniques called *dynamic execution*. The Pentium Pro, Celeron™, Pentium II, Pentium III processors are all based on the P6-microarchitecture.

To solve this problem, the microprocessor looks at more than the next two instructions when deciding which instructions to execute in parallel. The hardware solution is a complex, multi-stage design that effectively looks ahead at a window of instructions, about the next twenty, and selects up to three that can be executed in parallel. The three instructions selected generally are not three in a row, so the processor has to do a lot of work to keep track of the order and get the proper results. The effect is dramatic. Suddenly, the superscalar design that struggled to execute two instructions in parallel, if they happened to occur in the proper sequence, looks ahead to find instructions which can run in parallel. This technique is successful enough to often find three instructions to do in parallel.

The problems of dynamic execution are more complex than just looking ahead and grabbing instructions to run. Techniques like register renaming have to be used to break what we call false dependencies, which have to exist due to the small register set in the IA-32 processors. There are other complications as well; each of which had to be satisfied in order to make dynamic execution a reality.

EPIC avoids the need for such complex hardware. Parallelism is explicitly specified in the instructions instead of being something for which the processor must hunt to run programs fast. Dynamic execution became a necessity when the time came to run three instructions at once using an instruction set that started with microprocessors which ran one instruction at a time. EPIC starts with an architecture that can run six instructions at once, without employing superscalar detection hardware and dynamic execution.

EPIC—NEW FEATURES

EPIC is about parallelism. Unlike earlier parallel (VLIW) designs, EPIC does not use a fixed-width instruction encoding. Instead, instructions can be combined to operate in parallel from one to as many instructions as desired. To make this variable width work, one simple rule was imposed that earlier parallel designs did not impose. Programs must be written to work assuming *sequential semantics*

Consider a program to swap data:

```
TMP = A; A=B; B=TMP
```

In a computer with parallel semantics, a program can simply tell the machine to execute A=B and B=A in parallel. With EPIC, you may not do this because sequential semantics would not swap the data. That is, A=B and B=A in parallel means something different than doing A=B then B=A.

This simple restriction makes it possible to build machines with different levels of parallelism, to execute the same programs and get the same results. This *scalability* is another important characteristic of EPIC.

The rest of the new features in EPIC are designed simply to allow a compiler to produce code to use all this parallelism. The three key concepts to understand are:

- Speculation
- Predication (and Parallel Compares)
- Rotating register file

These capabilities make explicit parallelism useful. Without them, the parallelism would go unused so often that a very wide machine would never be busy enough to justify building it. Reviewing these concepts

early should enable you to appreciate the need for the many instructions documented in the remainder of the book.

SPECULATION

In a modern computer, loading data as early as possible is very useful for speeding up systems. If the data is not in cache when loaded by the program, execution must wait while the data is retrieved from memory. This delay is due to *memory latency*. This potential for delay leads to an obvious desire to do load instructions as early as possible, preferably so early that even waiting for memory is not a problem. Realistically, the compiler can try to eliminate at least some of the memory latency delays by moving LOADs earlier.

Several things limit how early one can speculate a load from memory, and have it be useful:

- Data is loaded so early that the register holding the data has to be used for something else before the program gets around to using the data that was loaded.
- Data is loaded before it is certain that the memory access would actually happen in the program (*control speculation*).
- Data is loaded before the necessary data has been computed and stored in memory (*data speculation*).

In most machines, these limitations severely impede our ability to do loads early enough to get dramatic speed-ups. With EPIC, each of these problems is addressed. Many more details are given in the chapters of this book, but here is a summary of the problems and their EPIC solutions.

Problem #1: Data is loaded so early that the register holding the data has to be used for something else before the program gets around to using the data we loaded.

Solution: The architecture provides lots of registers: 128 for integers data, 128 for floating point data, 64 for predicates (True / False).

Problem #2: Data is loaded before it is certain that the memory access would actually happen in the program. The problem here is that the load could cause a fault (possible program termination) in cases where a branch in program execution would have resulted in the LOAD never taking place.

Solution: Instead of using a normal load operation, the processor executes a speculative load (`ld.s`) instruction earlier in the instruction steam. The register uses a bit associated with it known as a *Not a Thing* (NaT) *bit* (NaTVal for floating-point registers) to keep track of whether or not the data is valid. Later, when the load is known to be necessary, a speculation check (`chk.s`) instruction checks the NaT bit to confirm that the data is still valid and if invalid data is detected initiates a recovery.

Problem #3: Data is loaded before the necessary data has been computed and stored in memory.

Solution: Instead of using a normal load, the processor executes an advanced load (`ld.a`) instruction earlier in the instruction stream. An entry in a special register called the *advanced load address table* (ALAT) marks the occurrence of an advanced load. Later, when the load is proven to be necessary, a check load (`ld.c`) instruction checks the entry in the ALAT to confirm that data is valid and to initiate a recovery if it is not.

Problem #4: Data is loaded both before it is certain that the memory is accessible and that no stores can overwrite the memory.

Solution: Use an advanced load (`ld.a`) instruction then an advanced load check (`chk.a`) instruction instead of a load check (`ld.c`). This solution is simply a combination of the concepts mentioned in the last two problems.

SPECULATING MORE THAN LOADS

Once the processor is loading data early, it is only natural to start doing computations on the data early too. Doing computations early simply means using the data from an advanced or speculative load before it can be certain that the data was really loaded, or that it was valid data.

By loading data early, and starting a chain of dependent computations (ones that use the data as input), the program can get work done earlier by using up the parallelism available in the processor. The consistent theme is using the parallelism available in EPIC to get work done even before the results are need, or before all the inputs are ready (the loaded data). The trick is to fill up the available parallelism with those loads and computations that are most likely to be needed. In this way, the processor can maximize use of the benefits that the extra parallelism from EPIC has to offer.

The problem with doing the computations early, based on loads that may not have gotten the right data, is what to do if we eventually determine that the loaded data is not valid. In the case of an advanced load, this mistake could mean that the wrong data was loaded because a store has overwritten that memory location and changed the data. For this case, the advanced load check identifies data that is invalid and initiates a branch to *recovery code*. Since more happened to the data than just loading it, the fix-up code needs to do more than just reload the data. It needs to redo the computations with the correct data. Recovery code is something we want to use only occasionally. In the normal case, we should get the benefits from speculating correctly and from getting the resulting load and computations done early.

PREDICATION

A predicate register controls practically every instruction defined by EPIC. Most instructions are written to make no use of predication. Instead, the compiler specifies a predicate register to control them that is always True. Therefore, instructions that appear to not be predicated in the assembly language are actually coded to use the predicate register, which is always True.

The use and benefits of predication were covered earlier. In summary, predication allows instructions to be executed in parallel, with some instructions turned on and some instructions turned off, but without need of branches. The elimination of branches is beneficial, and the compiler often can pack the resulting predicated code with non-predicated code. Greater compactness more than makes up for the fact that instructions with false predicates are using up available parallelism in the machine to do nothing.

PARALLEL COMPARES

Predication of instructions by predicate registers means that the processor needs to compute the True/False values to be placed in the predicate registers. Three potential problems are identified when instructions start using predication:

Problem #1: It is useful to have the complement of a predicate.

Solution: The compare instructions can write the result to one predicate register and the complement to another. This separation allows a test for A>B to place the value True in one predicate register where it is used to predicate the code to be executed when A is greater than B, and to place the value False in another register to predicate the code to be executed when A is not greater than B.

Problem #2: If your program is comparing data, at least one of which originated from an invalid speculative load, it would be useful to not execute any code based on either output predicate because the comparison is useless.

Solution: Compare instructions set both output predicate registers to False if either of its inputs is an invalid result of a speculative load.

Problem #3: Compare instructions can only compare two numbers at a time, which implies that computing the value of the expression ((A>B) OR (B>C) OR (C>D)) must be computed as follows:

1. (A>B), (B>C), and (C>D) in one clock,
2. combine two of these results in another, and
3. combine that result with the third comparison in a third clock.

To compute a predicate for this expression takes three clocks, after the values of A, B, C and D are known, and the action takes five instructions.

Solution: The architecture provides special *parallel compare instructions* that are an exception to the rule that multiple instructions cannot write the same register at the same time (in parallel). The trick is that these instructions come in two versions. One writes True or does nothing; the second writes FALSE or does nothing. Using parallel compares, the computation of the proper predicate for this example expression takes only four instructions and one clock after we know the values of A, B, C

and D, much better performance than we could expect without these instructions.

ROTATING REGISTER FILE

Another frequently used program structure that can adversely affect performance is the loop. EPIC provides a number of programming constructs that permit loops to be performed more efficiently. Combining these with the parallelism naturally leads one to do *software pipelining* in this architecture. Software pipelining is a technique that re-codes loops to execute from multiple iterations in parallel with each other. Software pipelining normally leads to the need for extra special-purpose code in front of a loop and after the loop. This code, called the prolog and epilogue, deals with priming the pipeline and draining the pipeline, respectively.

In other architectures, this additional code has made software pipelining profitable only if the loop is known to run long enough to make up for the overhead. This estimation of time turns out to be impractical for a compiler to determine. Generally, compilers only pipeline loops involving floating-point operations since these tend to loop many times and the compiler often knows the number of times. The number of iterations to complete a loop is called the trip count. Software pipelining is seldom applied to integer loops because the overhead of having extra code is often a factor in slowing a program due to greater use of the instruction cache. Software pipelining is also often not applied to loops with non-constant trip counts because the prologue and epilogue code is more complex to create.

EPIC uses an elegant solution called rotating registers to allow very compact representation of software pipelined loops. With EPIC, a compiler can produce software-pipelined loops without having to generate prologue or epilogue code. The resulting code is simpler and takes up less space. This simplification makes it more likely that software pipelining can speed up code. For this reason, the compiler can be more aggressive about using this technique without the penalties you would expect with other architectures.